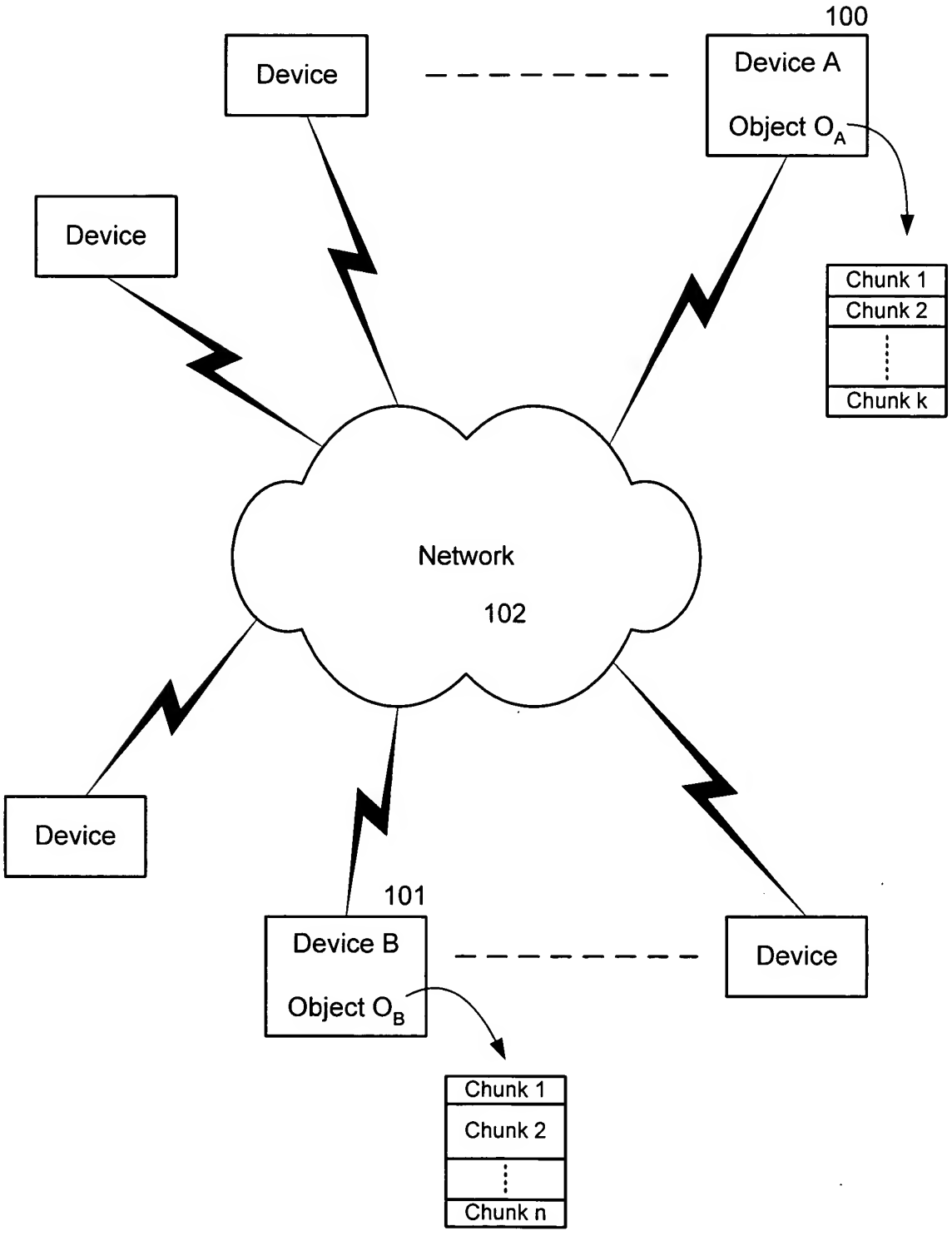
100
Device
Device A
Object O_A
Device
Chunk 1
Chunk 2
Chunk k
Network
102
Device
101
Device B
Object O_B
Device
Chunk 1
Chunk 2
Chunk n

FIG. 1

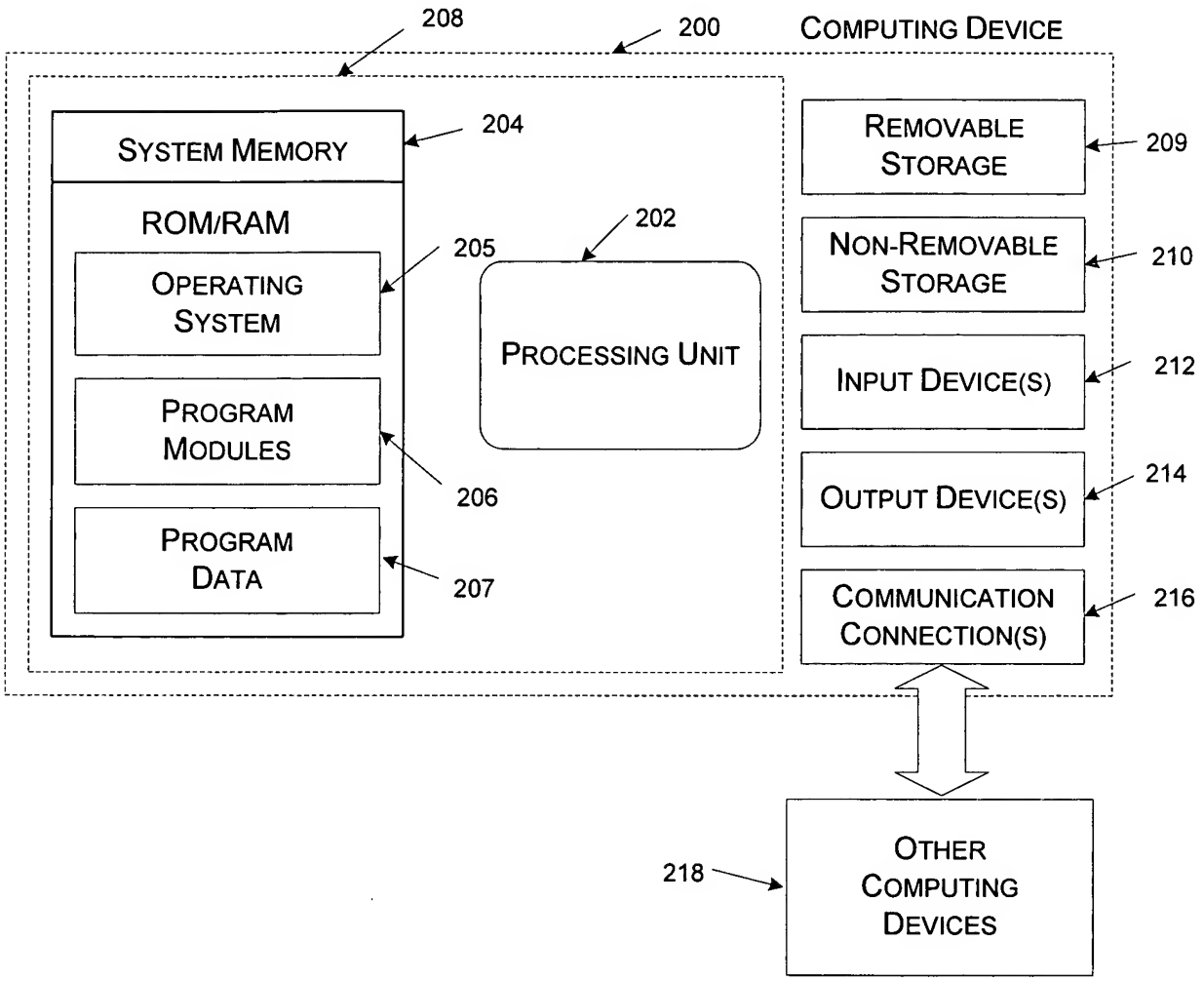
208
200
COMPUTING DEVICE
SYSTEM MEMORY
204
ROM/RAM
205
OPERATING SYSTEM
PROGRAM MODULES
206
PROGRAM DATA
207
202
PROCESSING UNIT
REMOVABLE STORAGE
209
NON-REMOVABLE STORAGE
210
INPUT DEVICE(S)
212
OUTPUT DEVICE(S)
214
COMMUNICATION CONNECTION(S)
216
218
OTHER COMPUTING DEVICES

FIG. 2

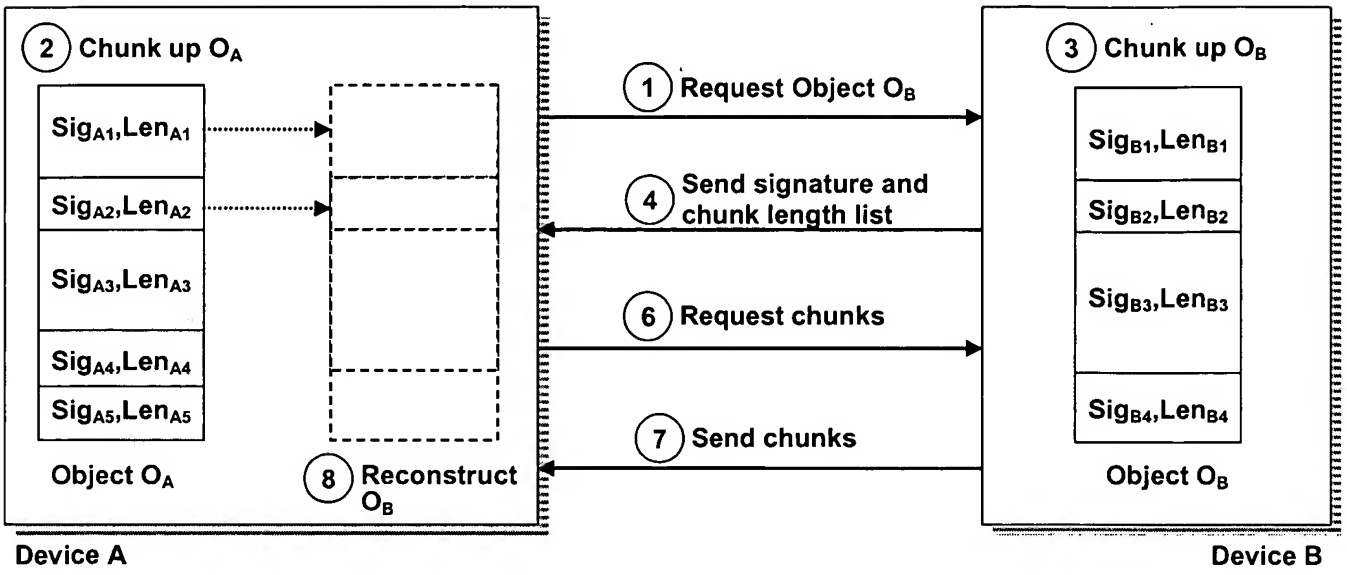
2 Chunk up O_A
Sig_{A1},Len_{A1}
Sig_{A2},Len_{A2}
Sig_{A3},Len_{A3}
Sig_{A4},Len_{A4}
Sig_{A5},Len_{A5}
Object O_A
8 Reconstruct O_B
Device A
1 Request Object O_B
4 Send signature and chunk length list
6 Request chunks
7 Send chunks
3 Chunk up O_B
Sig_{B1},Len_{B1}
Sig_{B2},Len_{B2}
Sig_{B3},Len_{B3}
Sig_{B4},Len_{B4}
Object O_B
Device B

FIG. 3A

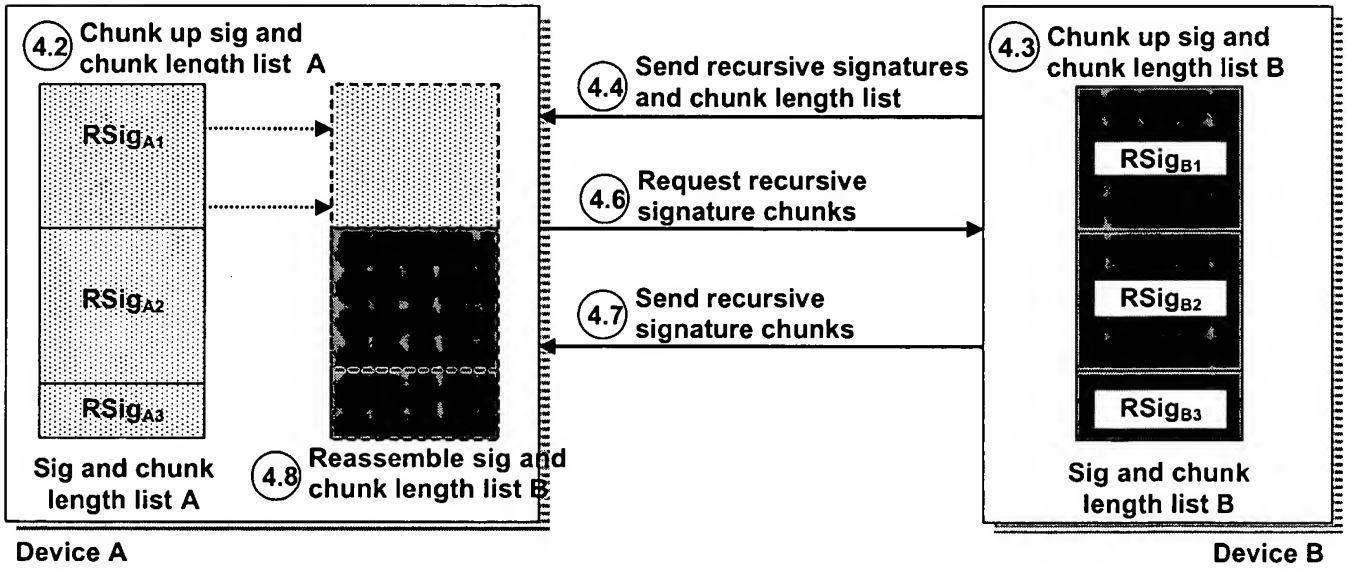
4.2 Chunk up sig and chunk length list A
$RSig_{A1}$
$RSig_{A2}$
$RSig_{A3}$
Sig and chunk length list A
4.8 Reassemble sig and chunk length list B
Device A
4.4 Send recursive signatures and chunk length list
4.6 Request recursive signature chunks
4.7 Send recursive signature chunks
4.3 Chunk up sig and chunk length list B
$RSig_{B1}$
$RSig_{B2}$
$RSig_{B3}$
Sig and chunk length list B
Device B

FIG. 3B

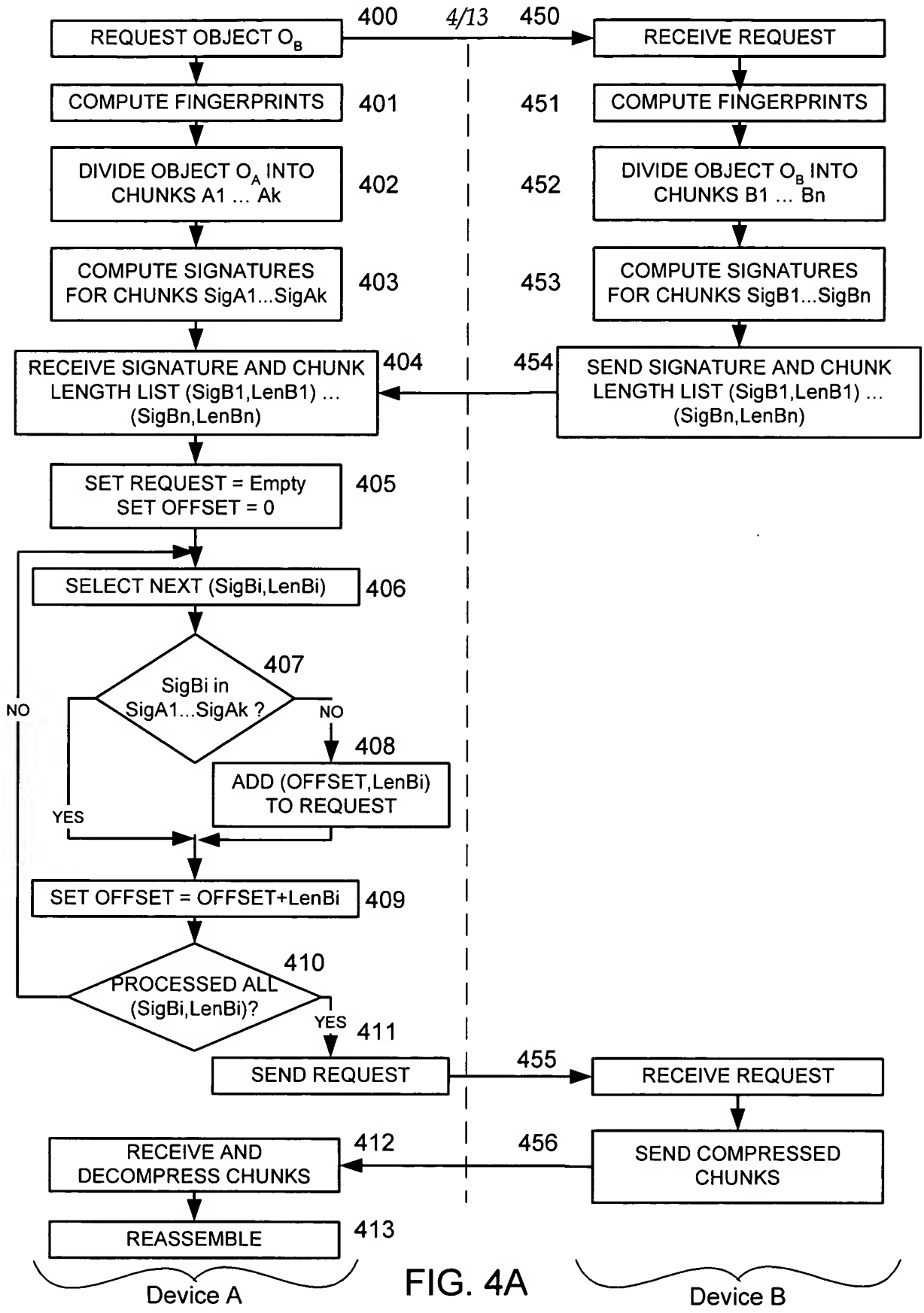
400
4/13
450
REQUEST OBJECT O_B
RECEIVE REQUEST
COMPUTE FINGERPRINTS
401
451
COMPUTE FINGERPRINTS
DIVIDE OBJECT O_A INTO CHUNKS A1 ... Ak
402
452
DIVIDE OBJECT O_B INTO CHUNKS B1 ... Bn
COMPUTE SIGNATURES FOR CHUNKS SigA1...SigAk
403
453
COMPUTE SIGNATURES FOR CHUNKS SigB1...SigBn
RECEIVE SIGNATURE AND CHUNK LENGTH LIST (SigB1,LenB1) ... (SigBn,LenBn)
404
454
SEND SIGNATURE AND CHUNK LENGTH LIST (SigB1,LenB1) ... (SigBn,LenBn)
SET REQUEST = Empty
SET OFFSET = 0
405
SELECT NEXT (SigBi,LenBi)
406
SigBi in SigA1...SigAk ?
407
NO
NO
YES
ADD (OFFSET,LenBi) TO REQUEST
408
SET OFFSET = OFFSET+LenBi
409
PROCESSED ALL (SigBi,LenBi)?
410
YES
411
SEND REQUEST
455
RECEIVE REQUEST
RECEIVE AND DECOMPRESS CHUNKS
412
456
SEND COMPRESSED CHUNKS
REASSEMBLE
413
Device A
Device B

FIG. 4A

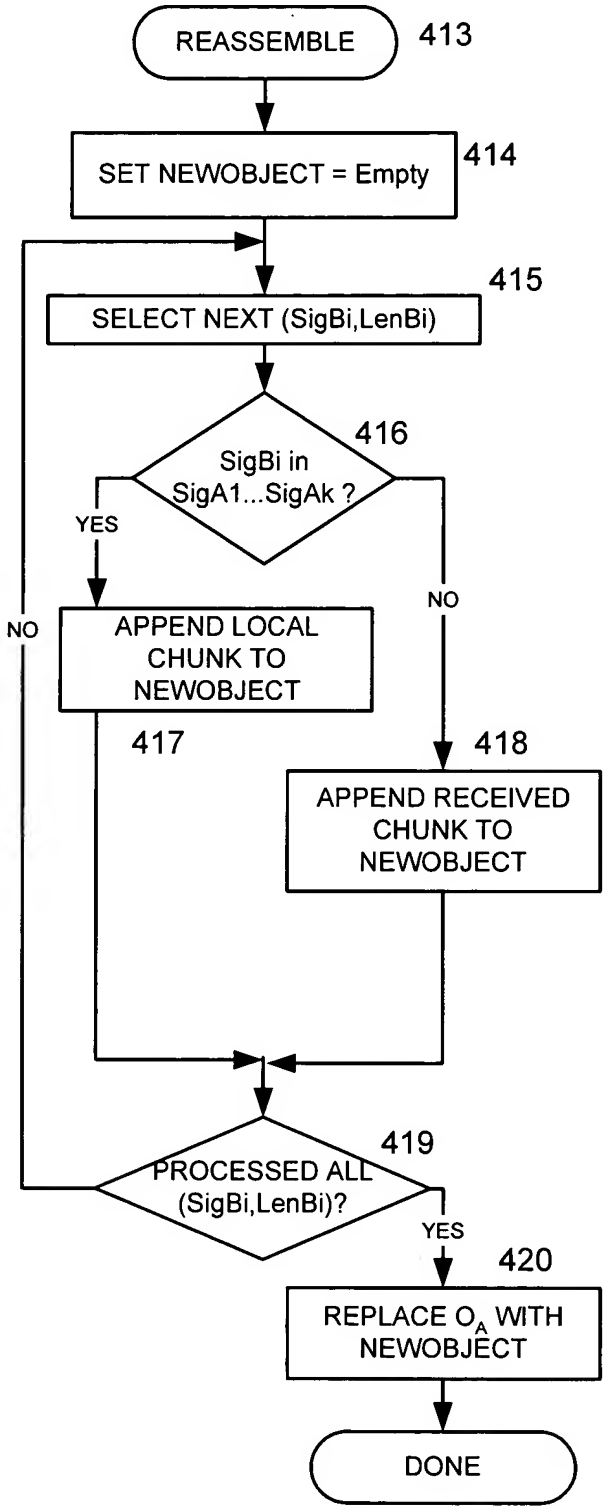
REASSEMBLE
413
SET NEWOBJECT = Empty
414
415
SELECT NEXT (SigBi,LenBi)
416
SigBi in SigA1...SigAk ?
YES
NO
NO
APPEND LOCAL CHUNK TO NEWOBJECT
417
418
APPEND RECEIVED CHUNK TO NEWOBJECT
419
PROCESSED ALL (SigBi,LenBi)?
YES
420
REPLACE O_A WITH NEWOBJECT
DONE

FIG. 4B

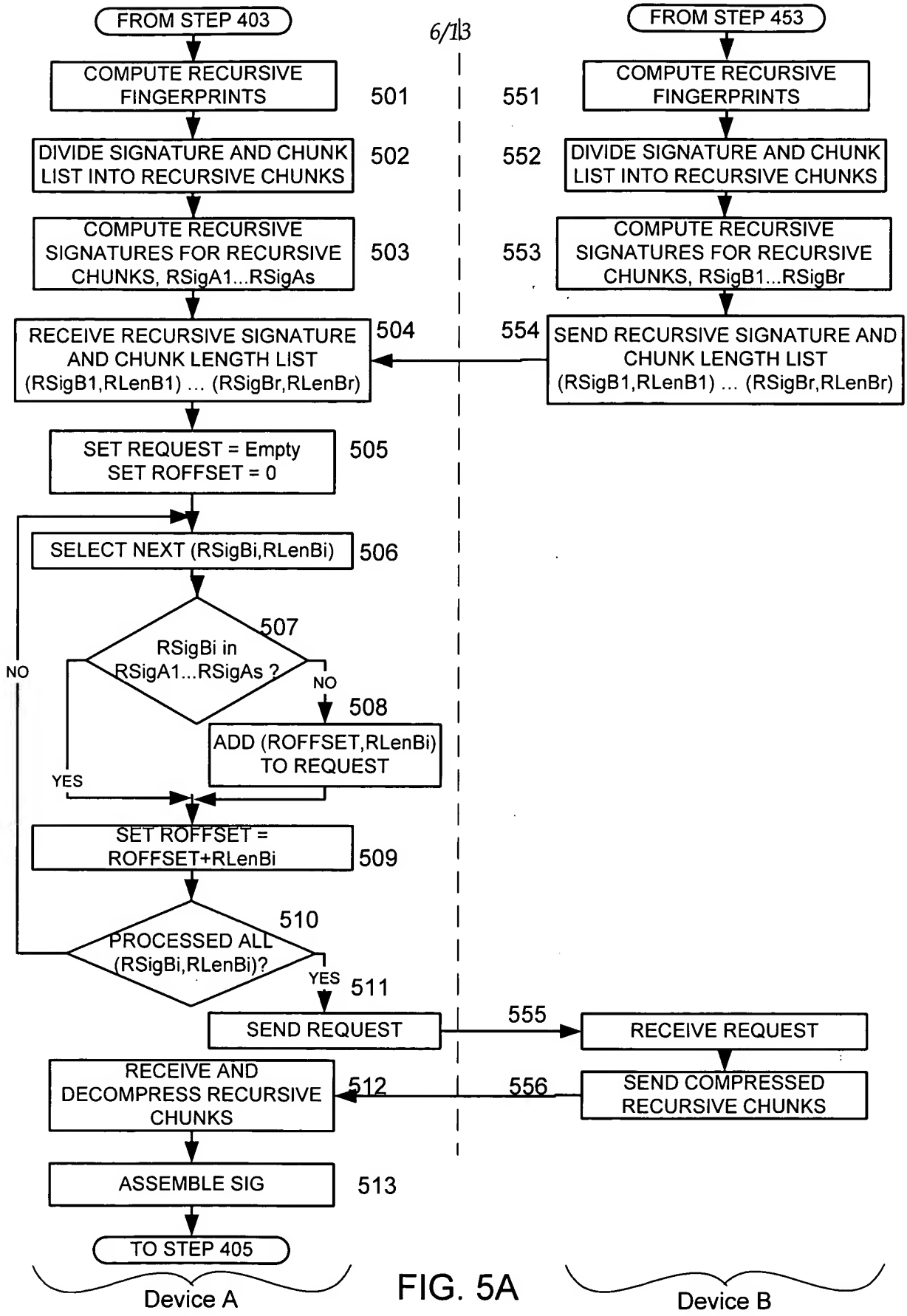
FROM STEP 403
COMPUTE RECURSIVE FINGERPRINTS
501
DIVIDE SIGNATURE AND CHUNK LIST INTO RECURSIVE CHUNKS
502
COMPUTE RECURSIVE SIGNATURES FOR RECURSIVE CHUNKS, RSigA1...RSigAs
503
RECEIVE RECURSIVE SIGNATURE AND CHUNK LENGTH LIST (RSigB1,RLenB1) ... (RSigBr,RLenBr)
504
SET REQUEST = Empty
SET ROFFSET = 0
505
SELECT NEXT (RSigBi,RLenBi)
506
RSigBi in RSigA1...RSigAs ?
507
NO
YES
NO
ADD (ROFFSET,RLenBi) TO REQUEST
508
SET ROFFSET = ROFFSET+RLenBi
509
PROCESSED ALL (RSigBi,RLenBi)?
510
YES
SEND REQUEST
511
RECEIVE AND DECOMPRESS RECURSIVE CHUNKS
512
ASSEMBLE SIG
513
TO STEP 405
Device A
FROM STEP 453
COMPUTE RECURSIVE FINGERPRINTS
551
DIVIDE SIGNATURE AND CHUNK LIST INTO RECURSIVE CHUNKS
552
COMPUTE RECURSIVE SIGNATURES FOR RECURSIVE CHUNKS, RSigB1...RSigBr
553
SEND RECURSIVE SIGNATURE AND CHUNK LENGTH LIST (RSigB1,RLenB1) ... (RSigBr,RLenBr)
554
555
RECEIVE REQUEST
556
SEND COMPRESSED RECURSIVE CHUNKS
Device B

FIG. 5A

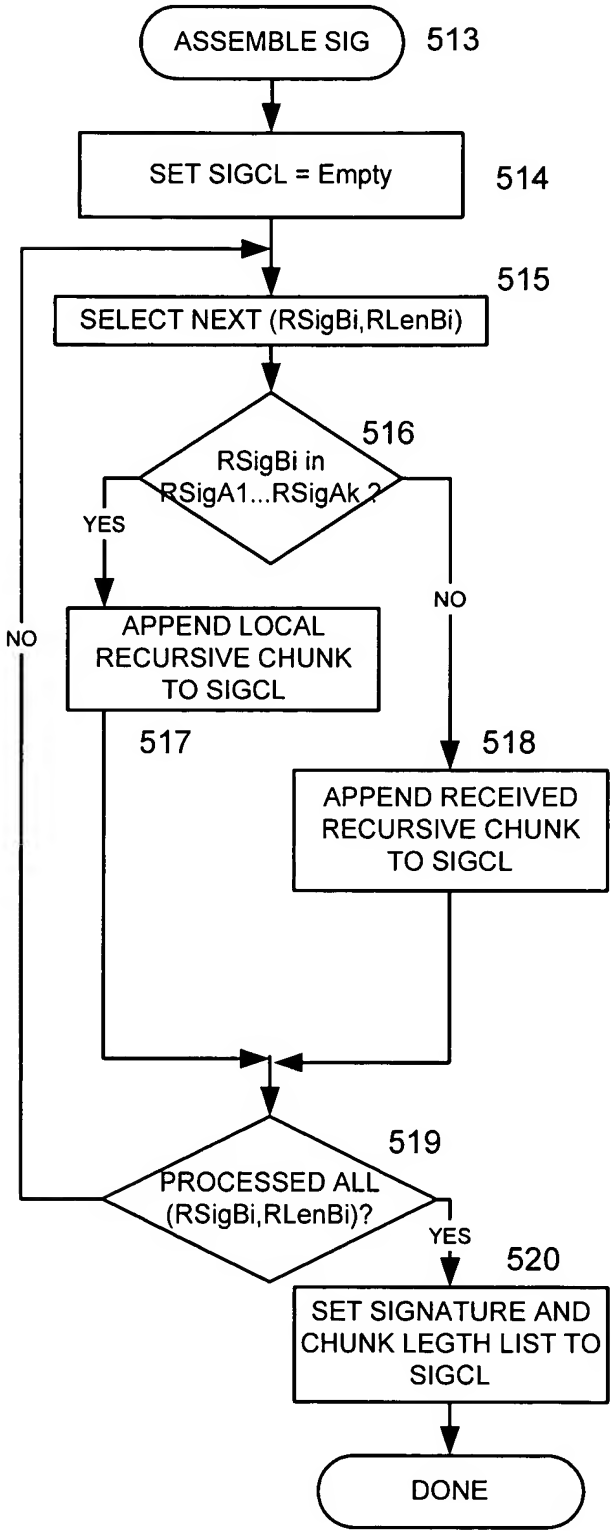
ASSEMBLE SIG
513
SET SIGCL = Empty
514
515
SELECT NEXT (RSigBi,RLenBi)
516
RSigBi in RSigA1...RSigAk ?
YES
NO
APPEND LOCAL RECURSIVE CHUNK TO SIGCL
517
NO
518
APPEND RECEIVED RECURSIVE CHUNK TO SIGCL
519
PROCESSED ALL (RSigBi,RLenBi)?
YES
520
SET SIGNATURE AND CHUNK LEGTH LIST TO SIGCL
DONE

FIG. 5B

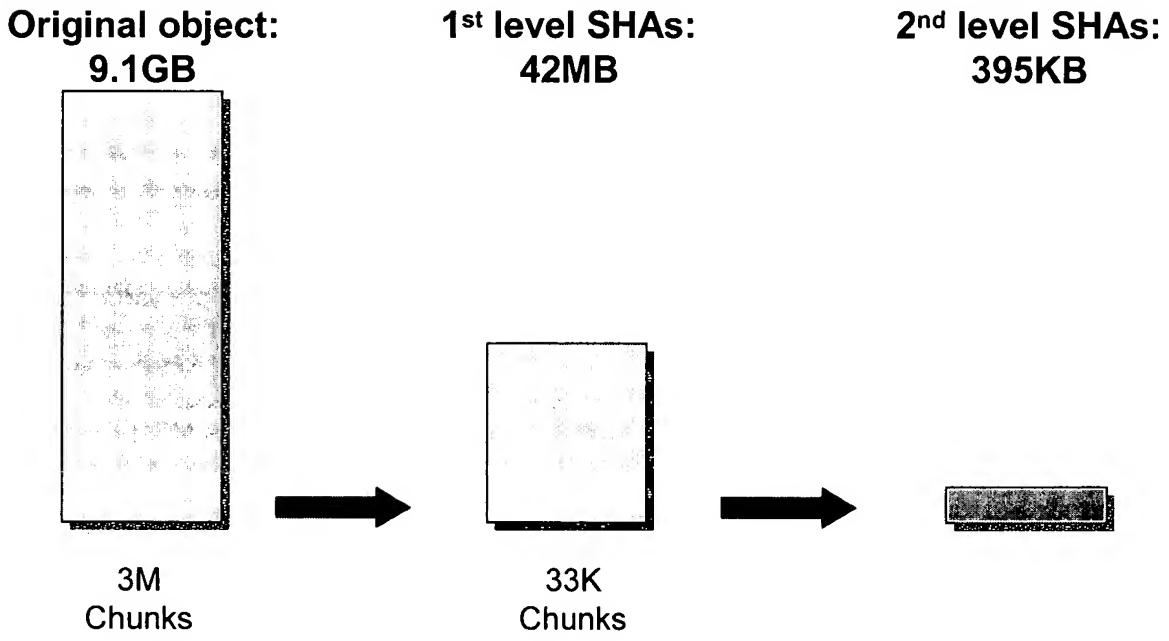
Original object:
9.1GB
1st level SHAs:
42MB
2nd level SHAs:
395KB
3M
Chunks
33K
Chunks

FIG. 6

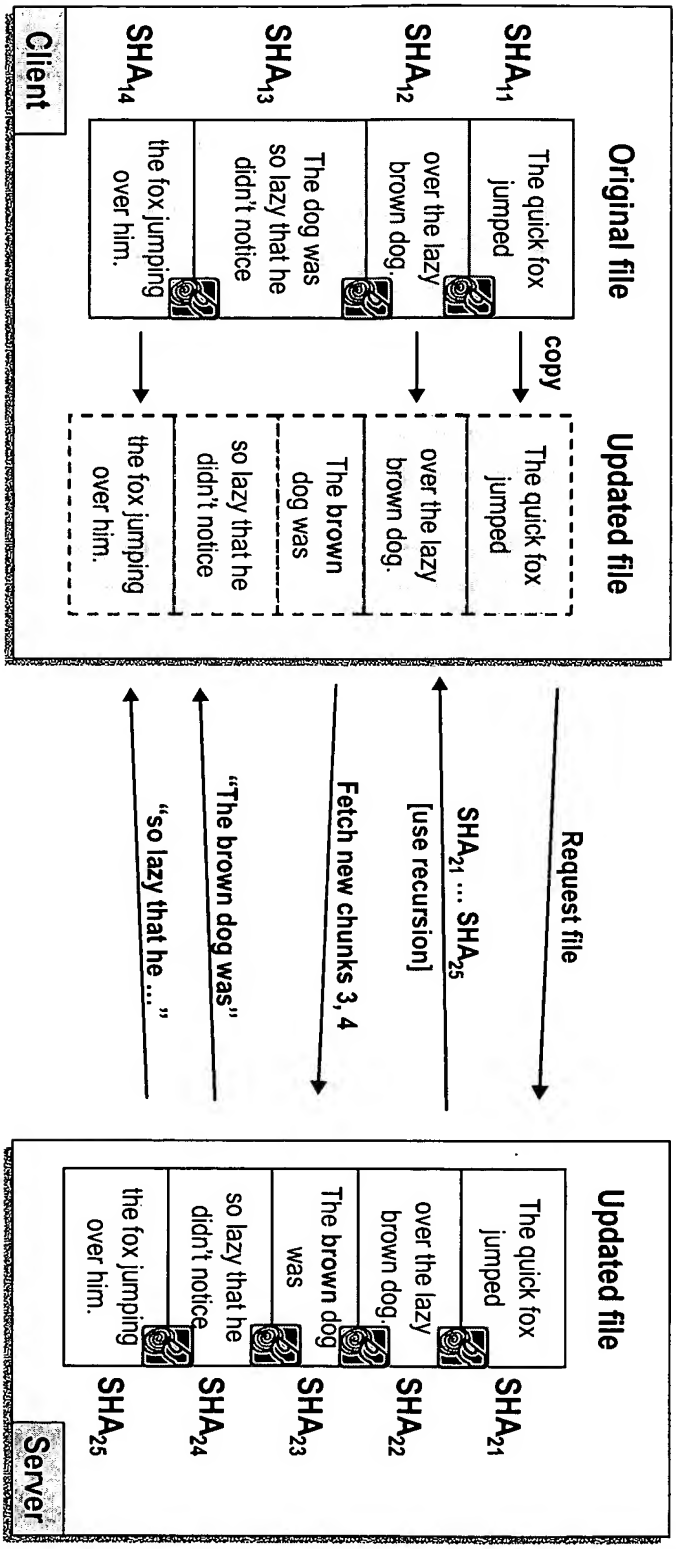
Client
Original file
SHA11
The quick fox jumped
SHA12
over the lazy brown dog.
SHA13
The dog was so lazy that he didn't notice
SHA14
the fox jumping over him.
copy
Updated file
The quick fox jumped
over the lazy brown dog.
The brown dog was
so lazy that he didn't notice
the fox jumping over him.
Request file
SHA21 ... SHA25
[use recursion]
Fetch new chunks 3, 4
"The brown dog was"
"so lazy that he ..."
Server
Updated file
SHA21
The quick fox jumped
SHA22
over the lazy brown dog.
SHA23
The brown dog was
SHA24
so lazy that he didn't notice
SHA25
the fox jumping over him.

FIG. 7

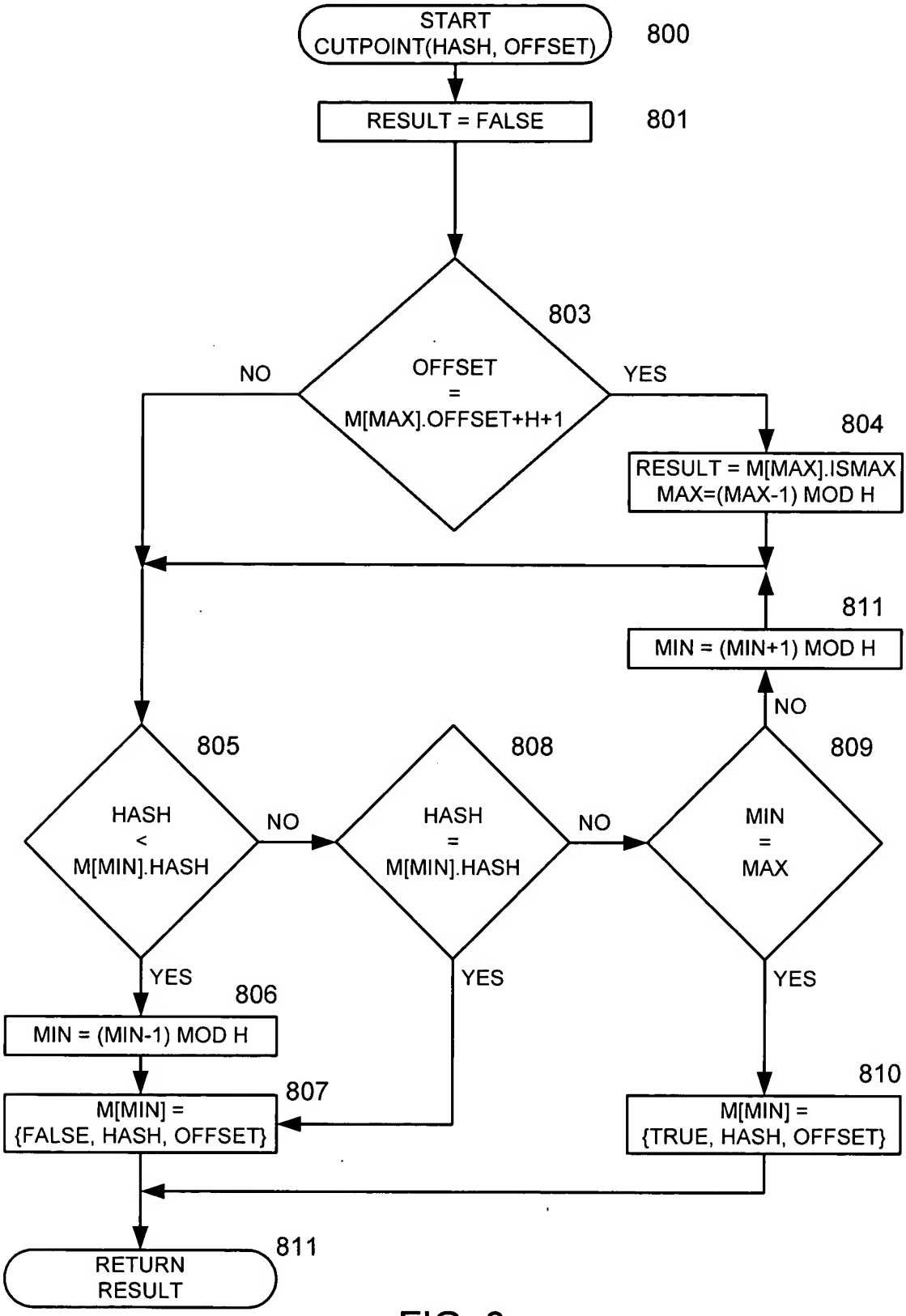
START
CUTPOINT(HASH, OFFSET)
800
RESULT = FALSE
801
803
OFFSET
=
M[MAX].OFFSET+H+1
NO
YES
804
RESULT = M[MAX].ISMAX
MAX=(MAX-1) MOD H
811
MIN = (MIN+1) MOD H
NO
805
HASH
<
M[MIN].HASH
NO
808
HASH
=
M[MIN].HASH
NO
809
MIN
=
MAX
YES
YES
YES
806
MIN = (MIN-1) MOD H
807
M[MIN] =
{FALSE, HASH, OFFSET}
810
M[MIN] =
{TRUE, HASH, OFFSET}
811
RETURN
RESULT

FIG. 8

```
structure Entry
  var isMax  as Boolean = false
  var hash   as Integer = 0
  var offset as Integer = 0

class LocalMaxCut
  h as Integer
  var min as Integer = 0
  var max as Integer = 0
  var M as Array of entry = new Entry[h]

  CutPoint(hash as Integer, offset as Integer) as Boolean
    var result = false
    step
        if M[max].offset + h + 1 = offset then
          result := M[max].isMax
          max := (max+1) mod h
    step
     while true do step
          step
            if M[min].hash > hash then
                step
                  min := (min-1) mod h
                step
                  M[min] := Entry(false, hash, offset)
                  return result
            if M[min].hash = hash then
                  M[min] := Entry(false, hash, offset)
                  return result
            if M[min].hash < hash and min = max then
                M[min] := Entry(true, hash, offset)
                return result
          step
                min := (min+1) mod h
```

FIG. 9

```
structure Entry
  var offset as Integer = 0
  var isMax  as Boolean = false
  var hash   as Integer = 0

class LocalMaxCut
  horizon as Integer
  var hashes as Seq of Integer
  var k as Integer = 0
  var l as Integer = 0
  var A as Array of Entry = new Entry[horizon]
  var B as Array of Entry = new Entry[horizon]

  CutPoints() as Seq of Integer
    var cuts as Seq of Integer = []
    for window = 0 to Length(hashes)/horizon do step
      let first = window*horizon
      let last = min((window+1)*horizon,Length(hashes)))-1
      cuts := cuts + CutPoint(first, last)
    return cuts

  CutPoint(first as Integer, last as Integer) as Seq of Integer
    step // Initialize A with the first entry at the offset
      k := 0
      A[0] := Entry(last,true,hashes[last])
      last := last - 1
    step // Update A[k] in the interval up to B[l]'s horizon
      while last > B[l].offset + horizon do step
        Insert(last)
        last := last - 1
    step // Update A[k] and B[l] in the remaining interval
      while last >= first do step
        Insert(last)
        if B[l].hash <= hashes[last] then
          B[l].isMax := false
        last := last -1
    step // determine whether A[k] is a cutpoint with respect to B
      A[k].isMax := A[k].isMax and
                    forall j in {0..l} holds
                    (B[j].offset + horizon < A[k].offset or
                     B[j].hash < A[k].hash)
    step // Set B to A for the next round and return cut-point
      B, l := A, k
      return if B[l].isMax then [B[l].offset] else []
```

FIG. 10

```
class LocalMaxCut
  Insert(offset as Integer)
    if hashes[offset] >= A[k].hash then
      if hashes[offset] = A[k].hash then
        // duplicated hashes within distance
        // of "horizon" are not maximal.
        A[k].isMax := false
      else
        A[k+1] := Entry(offset, true, hashes[offset])
        k := k + 1
```

FIG. 11